A Java Library for Fuzzy String Matching

Govinda Grings[1], John Healy[2]

[1] Dept. of Computing & Mathematics, Galway-Mayo Institute of Technology
govinda.grings@gmit.ie

[2] Dept. of Computing & Mathematics, Galway-Mayo Institute of Technology
john.healy@gmit.ie

Abstract

We describe the design and structure of a Java library for approximate string matching. The API provides a rich variety of flexible and extensible fuzzy data structures, including lists, tries and maps that support approximate string search and storage. Tests and benchmarks indicate a parity of space and time complexity with traditional, non-fuzzy, collections.

Keywords: Fuzzy String Matching, Approximate String Matching, Fuzzy Sets, Fuzzy Dictionaries.

1 Introduction

Approximate (or fuzzy) string matching considers the problem of finding a specific string within a search space, while allowing a given number of errors or deviations from a search pattern. This is a similar, but more complex, problem to that of finding exact matches for a given string within a search space, although the two problems share many concepts and methods.

Approximate string matching varies from exact string matching in that it can cope with corruptions in either the search pattern or the search space. These corruptions may arise from typographic mistakes, signal loss, minor variations in genetics, or a number of other factors, and may be considered to consist of the insertion, deletion, replacement or transposition of one or more characters between the search string and the string it is being matched against [1]. There are a number of fields where approximate string matching is useful, including text retrieval, computational biology and signal processing.

Methods for approximate string matching fall into two categories: on-line processing, where an algorithm searches through an entire search space from start to end looking for matches, and off-line processing, where a search space is indexed into some form of data structure which allows for faster matching at the cost of a significantly higher memory footprint [2].

In this paper, we present a Java utility library for indexed approximate string matching. The classes in this library are designed to be both flexible and extensible, and can be configured to use a variety of different storage algorithms. In addition to some of the more standard methods, such as lists and prefix trees, we also included some newer methods that utilise pre-hashing algorithms to sort index entries into groupings. The remainder of this discussion includes the methods and data structures used by the library and an overview of the design of the classes in the API. We then present the results of the various tests and benchmarks, used to compare the speed, efficiency and accuracy of the search methods implemented.

The source code for our fuzzy string-matching library is freely available from the project website at *http://sourceforge.net/projects/jfuzzystring/*.

2. Edit Distance & Fuzzy String Similarity

In order to make any kind of meaningful decision as to how closely two strings match, some type of algorithm must be employed to calculate the similarity between them. The most commonly used method is to calculate the edit distance between the two strings – the number of operations required to transform one string into another. Edit distance is typically computed using a dynamic programming algorithm, which reduces the string similarity problem into a number of easily manageable sub-problems. A matrix is generated to represent the edit distance between each part of the source and target strings. The value of each cell in the matrix is calculated by finding the minimum cost to reach that cell based on the values of adjacent cells and the costs associated with the operation to reach those cells. The final value in the matrix is the overall edit distance between the two strings [3, 4].

Hamming distance was one of the earliest approaches used to compute edit distance, and is calculated by counting the number of character substitutions needed to make two same-length strings equal [5]. This measure was expanded into the Levenshtein distance, which also includes the operations of insertion or deletion of characters, allowing an edit distance of non-equal-length strings to be computed [6]. These three operations, along with the transposition of two adjacent characters, account for approximately 80% of misspellings [1]. The edit distance including transpositions in addition to insertions, deletions and substitutions is usually referred to as the Damerau–Levenshtein distance.

In their simplest form, where only the number of required changes are counted, these algorithms still fail to take into account the distance of the errors. For example, given the word "LOTS", both the words "LOST" and "SLOT" have the same edit distance (one deletion and one insertion), despite the fact that "LOST" is clearly more similar. This problem can be overcome by assigning a different cost value to each operation, with transpositions having a lesser cost than an insertion and a deletion, so that a pair of words where only two letters need to be swapped are considered closer than a pair of words where one letter needs to be deleted from one place and added in another [7, 8].

Our library uses implementations of the above algorithms to calculate the edit distance between two words. Because our fuzzy dictionary uses values in the range [0.00…1.00] to represent the similarity of matches, the calculated edit distance must be converted into that range. This is done by calculating the cost of substituting every character in the first string (*s*) for every character in the second string, as a percentage, using the formula:

$$f = 1 - \left(\frac{d}{len(s) * c}\right) \qquad (1)$$

where *d* is the edit distance, *len(s)* is the number of characters in string *s* and *c* is the substitution cost. As the library loosely-couples its internal dictionary with a distance metric interface, it can be extended to include any string similarity algorithm capable of returning a fuzzy metric in the range [0.00…1.00]. Areas where our implementation could be improved include modifying it to only calculate a diagonal path through the matrix, rather than calculating every cell [9], and allowing more than one operation on each character, such as transposing two characters and then inserting new characters between them [10].

3. Dictionaries for Approximate String Matching

When considering which data structures were appropriate for inclusion in the library, we outlined two criteria:

- Every value stored in the dictionary must be unique, so as to allow additional information to be stored alongside the string itself.
- The data structure used in the dictionary should be object-oriented.

Using these criteria, a variety of fuzzy dictionary data structures were developed as part of the library and are outlined in the remainder of this section.

3.1 Simple List

The simplest method of indexed approximate string matching is to store the entire dictionary as a list. When searching, the entire list is iterated over and each entry is compared for similarity to the search string using a string similarity algorithm. While this is not a particularly efficient method, with a running time of $\Theta(n)$, it is completely reliable and its implementation is extremely straightforward.

3.2 Neighbourhood Generation with Exact Matching

There are only a finite number of possible strings that fall within a certain margin of error of any given string. Thus, when searching for approximate matches to a given input string, one approach is to generate all of the possible variations of the string that fall within a desired margin of error, and then perform an exact search on the index with each of the generated terms. However, as the length of the search string and the acceptable margin of error grow, the number of possible variations to search for increases exponentially. Given an alphabet of size k, a search term of length n, and an acceptable margin of error of e, the number of neighbours that need to be generated can be calculated as $(k \bullet (2n+1) + n - 1)^e$. This implies that neighbourhood generation is useful for small search strings and small margins of error, but it becomes decreasingly useful as the sizes involved increase [11].

Our implementation of this type of search allows for different neighbourhood generation algorithms to be swapped in freely. We provide a default algorithm that performs a given number of simple operations (insertions, deletions, substitutions and transpositions) to the search string, with insertions and substitutions limited to the 26 letters of the alphabet. In our tests, we used this algorithm with an operation limit of 2 changes to the search string. Each of these neighbours is then searched for using fast exact string matching operations on a *HashMap* containing the dictionary entries.

3.3 Approximate String Search in Tries

A trie, or prefix tree, uses a tree structure to hold the entries in a dictionary, organised by shared prefixes. The branch nodes in the tree each contain a part of a string that appears more than once, while the leaf nodes contain the remaining (unique) part of each string [12]. When an entry is added to the tree, any existing entries that share a prefix with the new entry will be split into a branch node and a leaf node, and the new entry will be added under that branch. For example, if the trie contains the word “garlic” and the word “garden” is added, a new branch node “gar” will be added, and both “lic” and “den” will be added as leaf nodes under that branch.

Our implementation of trie transversal for a key search uses a variant of a breadth-first search. As each node on a search path is evaluated, the distance to that node is calculated. If the distance is greater than the allowed margin of error, then that branch of the index is discarded. In this manner, all non-matching paths through the graph will be trimmed, and the search will finally return all the remaining

leaf nodes that fall within the desired margin of error [13]. The algorithm will initially need to examine a very large number of paths, as it needs to follow all paths to a depth equal to the given error margin before they can be eliminated; however, once this depth is reached, the remaining number of nodes to be examined decreases significantly.

3.4 Hash Maps & Approximate Key Search

Hash maps are dictionary data structures that use a key and hashing function to provide rapid, $\Theta(1)$, access to a set of mapped values. The rapid access time of hash maps is accomplished by transforming a key value into an integer that corresponds to a table index. The hashing function should ideally be very simple, so that it can be calculated quickly, but should also ensure that two distinct keys will be assigned distinct indices. The latter is not always possible; when the hashing function computes the same index for two distinct keys, the hash map stores both values linked to the same index. In this event, the retrieval algorithm will need to evaluate both of the keys stored in that index for equality with the search string in order to determine which one is the correct match. This is called a collision, a situation that hashing functions are generally designed to keep to a minimum [14].

In a hash data structure, the hash key is used to functionally determine a mapped value. The implication of this property is that, although redundancy is permitted among the values in a hash map, the hash keys must be unique. While this uniqueness requirement provides hash structures with the underlying property to facilitate speed, it constrains access to exact matches of keys. A recently published alternative, called a *Fuzzy Hash Map* [15], is capable of supporting approximate matching of hash keys, including strings. Fuzzy hash maps encourage controlled collisions to place strings into groups or categorisations, based on a pre-hashing algorithm that determines which group each string belongs to. The same pre-hashing algorithm can then be applied to a search term in order to determine the subset of the search space to be searched. The major drawback of this approach is that, as the pre-hashing algorithm sorts each item in the structure into one specific group. Thus, two words which have a low edit distance but are not sorted into the same group will not be considered to match.

Instead of using the internal mechanics of the *HashMap* directly, our implementation uses a *HashMap* to map each of the pre-hashed values with a different sub-dictionary. This achieves the same effect of dividing the search space into groups controlled by a pre-hashing algorithm, while allowing the flexibility of different types of sub-dictionary implementation and the ability to return sets of related results within a given search threshold. Three distinct pre-hashing approaches are implemented in the library. The default implementation used by our library is a pattern-based pre-hashing algorithm that can be configured by a pattern string. Hash (#) characters in the pattern are included in the pre-hash, while any other characters are ignored.

We also experimented with using a pre-hashing algorithm that uses neighbourhood generation to assign each entry in the dictionary to multiple groupings with different variations on the pre-hashed string, and using the pre-hash directly to find the appropriate grouping for a search term. Even with a simple neighbourhood generation algorithm of substituting one letter in the search string for another, this lead to a hundred-fold increase in dictionary size – every possible change of one character in a 4-character pre-hashed string results in 104 unique strings – so this method was deemed to be impractical for actual usage. In our experiments, the memory required to hold this dictionary was phenomenal, and as such we decided to exclude it from testing. The pre-hashing algorithm is still, however, included in our library.

Given a search string and a maximum number of errors that is less than the length of the string, it is reasonable to assume that there is some subsequence of the search string that does not contain any errors. The concept of filtering is to divide a search pattern into smaller sub-patterns, and then search for exact matches of each of these sub-patterns in an index. The area around each match is then examined using approximate matching to determine whether it is an acceptable match for the entire search pattern [16]. We adapted this concept into a pre-hashing algorithm to use with our hashed

dictionary. This algorithm divides up each term into every possible sub-string of a given length, and then assigns the entry to multiple groupings, one for each of the sub-strings. This method is similar to neighbourhood generation, but is significantly more efficient than brute-force neighbourhood generation, as groupings are only created for valid sub-strings of words. As such, each entry in the dictionary is added to merely $1 = l - f$ groupings, where l is the length of the string and f is the size of the sub strings produced by the filtering algorithm.

4. API Class Design

One of our primary goals when creating the approximate string-matching library was to make it as flexible and as easily extensible as possible. While everything in the library is based around the approximate matching of strings, our implementation uses generics for the dictionary implementations themselves, allowing them to store any type of Java object. Because all Java objects expose the *toString()* method, this is used internally to obtain a string representation of the object for approximate matching. Although the library is primarily designed for use with String objects, the use of generics adds extra flexibility and enables other types to be manipulated in an approximate manner, so long as they can provide appropriate string representations of themselves.

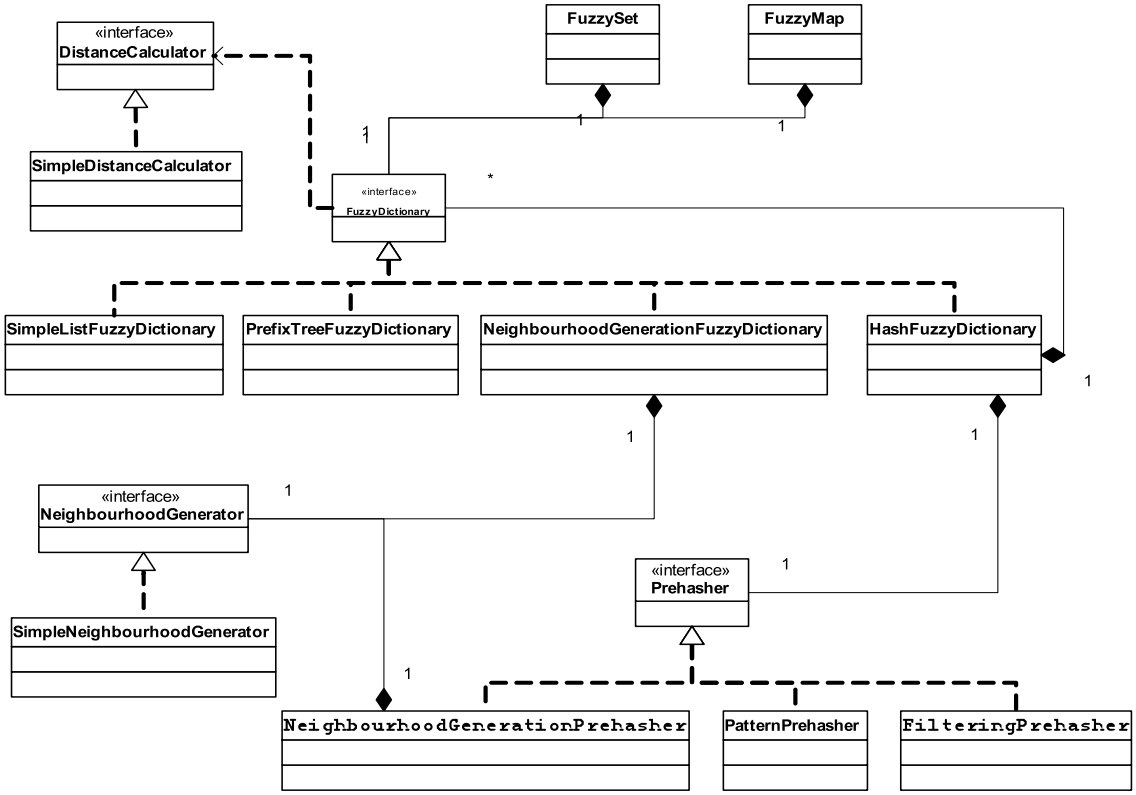

Figure 1: UML diagram depicting the relationships between the primary classes in the library.

At the heart of the system is the *FuzzyDictionary* interface, which defines the operations that all dictionary implementations can perform, such as adding and removing words and performing fuzzy searches. The distance calculation algorithm to be used by any implementation can be specified at run time using the *DistanceCalculator* interface. Specific types of dictionary implementations also use other sub-components, specifiable through interfaces: *NeighbouhoodGenerationFuzzyDictionary* uses a configurable type of *NeighbourhoodGenerator*, while the *HashFuzzyDictionary* can be configured to use different *Prehasher* implementations.

The library also includes two high-level fuzzy data storage classes: *FuzzySet* and *FuzzyMap*. Both use configurable *FuzzyDictionary* types as their underlying data structure, but can be used without worrying about the specific implementations. *FuzzyMap* stores key-value pairs, where the keys are fuzzy strings and are matched using fuzzy matching algorithms, and the value can be any type of Java object. *FuzzySet* is simply a wrapper for a *FuzzyDictionary* implementation that allows itself to be queried as to whether the dictionary contains a specific value, and returns the similarity of the closest match to the string.

5. Benchmarking & Results

To benchmark the fuzzy data structures in the library, we created a suite of automated tests to run on each of the algorithms. Each dictionary implementation was tested, by first loading a large library of words into it, and then querying it with sets of test words of different lengths and with different numbers of errors. The set of words was taken from the English dictionary files of the ISpell Unix spell checker [17] and consists of 144,105 unique words. We randomly selected 50 words from the word list in each of the following three categories:

- Short words: 4 characters in length
- Medium words: 5-8 characters in length
- Long words: 9+ characters in length.

We then developed test sets for each of these words with differing numbers of changes, i.e. insertions, deletions, substitutions and transpositions of characters. Figure 2 shows the average number of comparisons made during the searches in the various test sets. Each comparison involves the calculation of the distance between the search string and an entry in the dictionary.

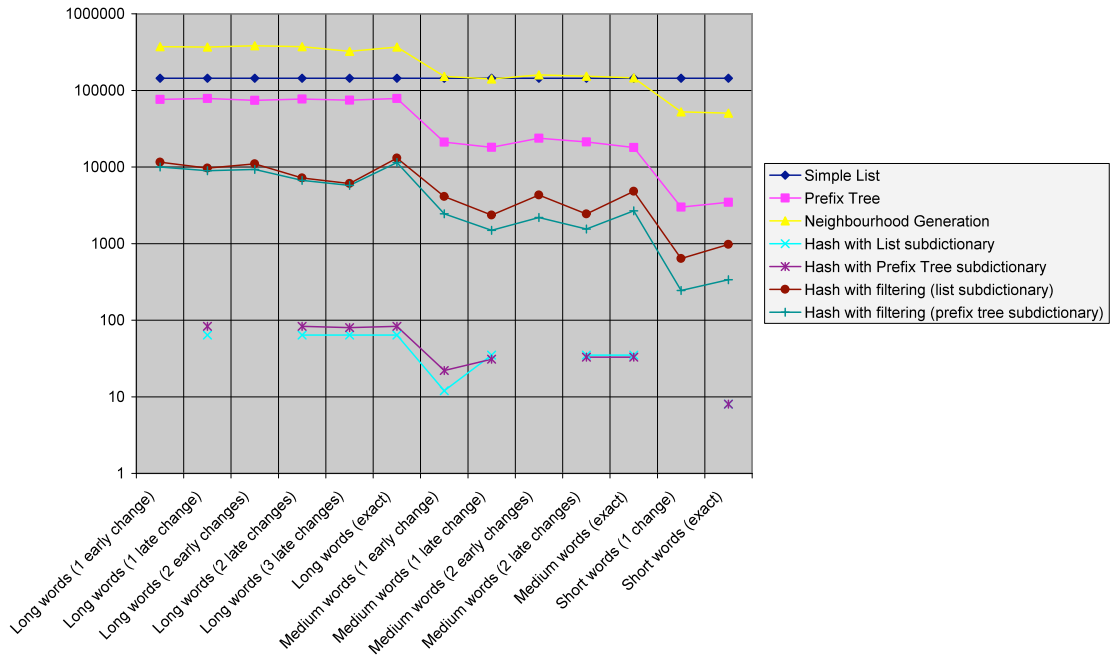

Figure 2: Comparisons made (logarithmic scale; lower is better).

Figure 3 shows the average time required to find each word in the test set, in microseconds. Note that some of the data points are missing for the non-filtering hashed algorithms. This is due to them being unable to find any matches in the given test set.

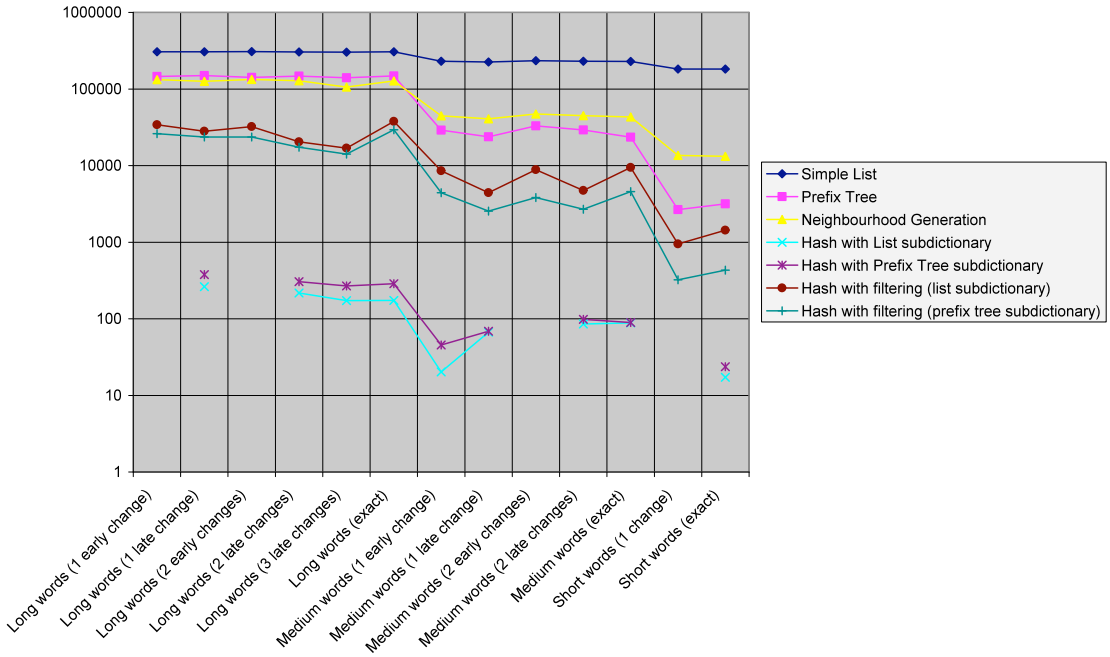

Figure 3: Average search times in µsec (logarithmic scale; lower is better).

Figure 4 shows the overall average search time for all test sets, ignoring match failures.

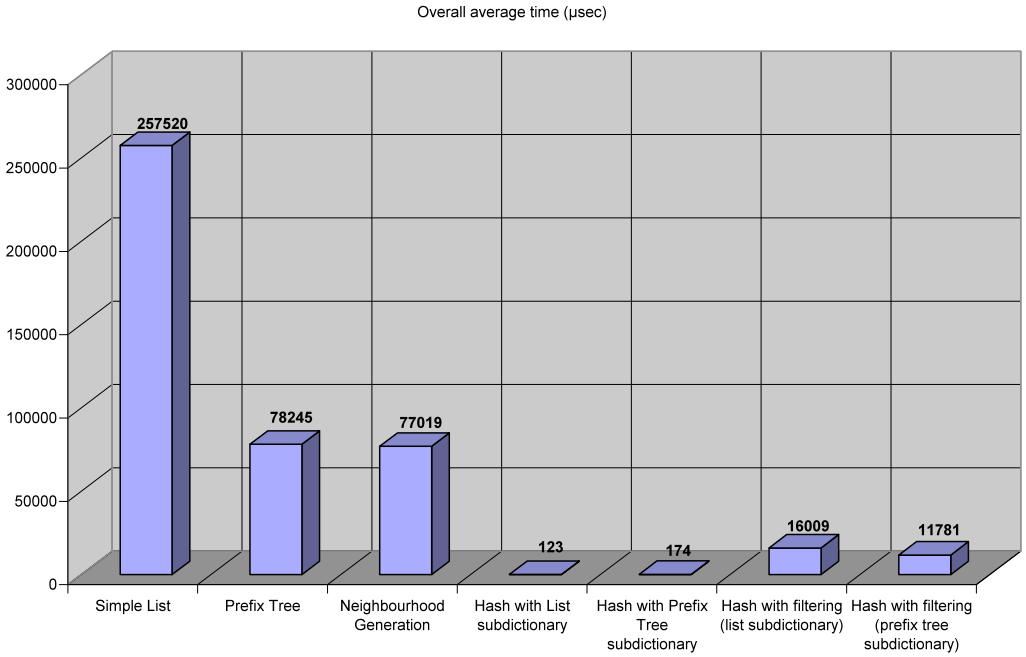

Figure 4: Overall average search times in µsec for all sets (lower is better).

The non-filtering hash algorithms are clearly orders of magnitude faster than any other algorithm, provided that both the search term and the word being sought share the same pre-hashed value. The filtering hash algorithms, while being significantly slower than the non-filtering ones, are still much faster than any other algorithm, while maintaining a high degree of accuracy. This is consistent with the time complexity of exact string matching and demonstrates that using a fuzzy approach does not have an adverse impact on running time.

6. Conclusions

In this paper, we presented an open-source Java library for fuzzy string matching which is flexible and configurable with a number of different dictionary implementations. We then described some of the tests used to benchmark the performance and accuracy of each of the implementations in relation to one another.

These tests clearly show that the use of a pre-hashing algorithm to group words into a number of smaller sub-dictionaries massively reduces the time required to find matching records, with the limitation of being unable to find records where the search string does not compute the same pre-hash as the desired entry. While using neighbourhood generation to assign each entry to multiple grouping proved to be completely impractical in terms of memory usage, the use of a filtering algorithm for the same purpose proved to be extremely effective at reducing the number of match failures, without losing too much of the speed advantage gained by using pre-hashed groupings. However, the filtered dictionaries use significantly more memory than the other algorithms.

The combination of hashed grouping and filtering provides an extremely viable, efficient and effective approximate string matching dictionary that should be suitable for most approximate string matching purposes, and we hope to further develop the library in the future.

References

[1] Damerau, F.J. (1964). *A technique for computer detection and correction of spelling errors. Communications* of the ACM, 7(3), pp.171–176.

[2] Navarro, G. (2001). *A guided tour to approximate string matching.* ACM Computing Surveys (CSUR), 33, pp.31–88.

[3] Wagner, R.A. & Fischer, M.J. (1974). *The string-to-string correction problem.* Journal of the ACM (JACM), 21(1), pp.168–173.

[4] Gilleland, M. (2008). Levenshtein Distance. Available at: http://www.merriampark.com/ld.htm

[5] Hamming, R.W. (1950). *Error detecting and error correcting codes.*

[6] Levenshtein, V.I. (1966). *Binary codes capable of correcting deletions, insertions, and reversals.* In Soviet Physics Doklady. pp. 707–710.

[7] Bookstein, A., Tomi Klein, S. & Raita, T. (2001). *Fuzzy Hamming Distance: A New Dissimilarity Measure (Extended Abstract).* In Combinatorial Pattern Matching. pp. 86–97.

[8] Bookstein, A., Kulyukin, V.A. & Raita, T. (2002). *Generalized hamming distance.* Information Retrieval, 5(4), pp.353–375.

[9] Ukkonen, E. (1985). *Algorithms for approximate string matching.* Information and control, 64(1-3), pp.100–118.

[10] Lowrance, R. & Wagner, R.A. (1975). *An extension of the string-to-string correction problem.* J. Assoc. Comput. Mach, 22(2), pp.177–183.

[11] Hall, P.A.V. & Dowling, G.R. (1980). *Approximate String Matching.* ACM Computing Surveys (CSUR), 12, pp.381–402.

[12] Fredkin, E. (1960). *Trie memory.* Communications of the ACM, 3(9), pp.490–499.

[13] Ukkonen, E. (1993). *Approximate string-matching over suffix trees.* In Combinatorial Pattern Matching. pp. 228–242.

[14] Weiss, M.A. (1992). *Data Structures and Algorithms*, Benjamin/Cummings.

[15] Topac, V. (2010). *Efficient fuzzy search enabled hash map*. Soft Computing Applications (SOFA), 2010 4th International Workshop on. pp. 39–44.

[16] Takaoka, T. (1994). *Approximate pattern matching with samples.* Algorithms and Computation, pp.234–242.

[17] Atkinson, Kevin (2011). *Kevin's Word List Page*. Available at: http://wordlist.sourceforge.net/.